MTH 3220 R Notes 5

6 Data Frames

6.1 Creating and Viewing Data Frames

- ***Data frames***, like matrices, are two-dimensional data structures (they have rows and columns), but unlike matrices they can store data that are *heterogeneous* (i.e. that contain a mix of numeric *and* "`character`" data).

  For example, here's a data set that would be stored as a *data frame* in R:

Var1	Var2	Var3
Low	6	22
Low	3	18
Low	11	27
Med	9	27
Med	14	26
Med	15	29
High	19	28
High	18	31
High	21	30

- Each column contains values of a ***variable***, and each row corresponds to an individual on which those variables were observed. An entire row is called a ***multivariate observation***.

6.1.1 Creating Data Frames Using `data.frame()`

- We can create a data frame on the R command line using:

```
data.frame()       # Create a data frame from a set of vectors of the
                   # same length
```

  (We'll see later how to create a data frame by reading the data from a file.)

- In addition, several functions let us view various aspects of a data frame:

```
head(); tail()   # Prints the first (or last) six rows of a data frame
nrow(); ncol()   # Indicates the number of rows (or columns)
dim()            # Gives the dimensions (number of rows and columns)
str()            # Gives the structure of a data frame
is.data.frame()  # Indicates whether or not an object is a data frame
```

- Consider again the data set from above. After creating vectors containing the data:

```
v1 <- c("Low", "Low", "Low", "Med", "Med", "Med", "High", "High", "High")
v2 <- c(6, 3, 11, 9, 14, 15, 19, 18, 21)
v3 <- c(22, 18, 27, 27, 26, 29, 28, 31, 30)
```

we create a data data frame named `my.data` by typing:

```
my.data <- data.frame(Var1 = v1, Var2 = v2, Var3 = v3,
                      stringsAsFactors = FALSE)
```

Here's the result:

```
my.data

##   Var1 Var2 Var3
## 1  Low    6   22
## 2  Low    3   18
## 3  Low   11   27
## 4  Med    9   27
## 5  Med   14   26
## 6  Med   15   29
## 7 High   19   28
## 8 High   18   31
## 9 High   21   30

is.data.frame(my.data)

## [1] TRUE
```

Specifying `stringsAsFactors = FALSE` indicates that we *don't* want the `"character"` vector `v1` to be converted to a so-called *factor* when the data frame is created. The default for `stringsAsFactors` is `TRUE`. We'll see later what *factors* are.

- The functions `head()` and `tail()` print just the first and last six rows, respectively:

```
head(my.data)

##   Var1 Var2 Var3
## 1  Low    6   22
## 2  Low    3   18
## 3  Low   11   27
## 4  Med    9   27
## 5  Med   14   26
## 6  Med   15   29
```

They're useful with large data sets.

- To find out how many rows a data frame has, use `nrow()`:

```
nrow(my.data)

## [1] 9
```

6.1.2 Creating Data Frames by Reading Data from a File Using `read.table()` and `read.csv()`

- We can create a data frame by reading the data from a file using:

```
read.table()        # Read data from a text (.txt) file into a data
                    # frame
read.csv()          # Read data from a 'comma separated value' (.csv)
```

```
                    # file into a data frame
```

- Suppose we have a text (.txt) file, **C:\micedatafile.txt**, that contains the following data on mice:

Color	Weight	Length
Purple	23	3.8
Yellow	21	3.7
Red	18	3.0
Brown	26	3.4
Green	25	3.4
Purple	22	3.1
Red	26	3.5
Purple	19	3.2

- We read it into a data frame named `mice` by typing:

```
mice <- read.table("C:/micedatafile.txt", header = TRUE,
                   stringsAsFactors = FALSE)
mice
```

```
##    Color Weight Length
## 1 Purple     23    3.8
## 2 Yellow     21    3.7
## 3    Red     18    3.0
## 4  Brown     26    3.4
## 5  Green     25    3.4
## 6 Purple     22    3.1
## 7    Red     26    3.5
## 8 Purple     19    3.2
```

- Here are some comments about using `read.table()`:
  - The usual back slashes (\) are written as forward slashes (/) in `read.table()`.
  - Specifying `header = TRUE` is used to indicate that the first row of the text file contains the variable names.
  - You can use either single (') or double (") quotations when specifying the file location and name.
  - By default, R recognizes one or more white spaces, tabs, or newline characters in the input file as separators of data values. Other separators can be specified via the `sep` argument.
  - As before, specifying `stringsAsFactors = FALSE` indicates that we *don't* want the character column (Color) to be converted to a *factor*.
  - `read.csv()` does the same thing as `read.table()`, but has different default values for some of the arguments.
  - To read data from an Excel file, either:
    * Save the Excel file as a tab delimited text (.txt) file, and then use `read.table()`.
      or
    * Save the Excel file as a 'comma separated value' (.csv) file, and then read it into R using `read.csv()`.

Section 6.1 Exercises

Exercise 1 Here are the mice data as three vectors:

```
col <- c("Purple", "Yellow", "Red", "Brown", "Green", "Purple", "Red",
         "Purple")
wt <- c(23, 21, 12, 26, 25, 22, 26, 19)
len <- c(3.8, 3.7, 3.0, 3.4, 3.4, 3.1, 3.5, 3.2)
```

After creating the vectors, write a command involving `data.frame()` that creates a data frame containing the data. Make sure the column names are `Color`, `Weight`, and `Length`.

6.2 Accessing and Replacing Elements, Rows, or Columns of a Data Frame

- Data frames have features of both matrices and lists. To extract a specific element, row, or column, we use:

```
[ , ]      # Access data frame elements, rows, or columns via their
           # row and column indices (separated by a comma)
[[ ]]      # Access a data frame variable (column) by specifying its
           # index or name
$          # Access a data frame variable (column) by specifying its
           # name
```

6.2.1 Accessing Rows and Columns Using `[ ]`

- As with matrices, we can access a specific element, row, or column of a data frame using single square brackets `[ ]`. For example, the value in the 3rd row and 2nd column of `mice` (created above) is:

```
mice[3, 2]

## [1] 18
```

The entire second column is accessed via:

```
mice[, 2]

## [1] 23 21 18 26 25 22 26 19
```

and the entire third row would be accessed via `mice[3, ]`.

- We can use square brackets and the assignment operator to *replace* a value in a data frame, for example:

```
mice[3, 2] <- 12
```

6.2.2 Accessing Columns Using `[[ ]]` and `$`

- In fact, **data frames are lists**:

```
is.data.frame(mice)

## [1] TRUE

is.list(mice)

## [1] TRUE
```

The list elements are the columns of the data frame.

- Therefore, the list operators `$` and `[[ ]]` can also be used to access columns of a data frame:

```
mice$Color           # Access a variable by name

## [1] "Purple" "Yellow" "Red"    "Brown"  "Green"  "Purple" "Red"    "Purple"
```

```
mice[[1]]            # Access a variable by list index (i.e. column number).
                     # We could also use mice[["Color"]].

## [1] "Purple" "Yellow" "Red"    "Brown"  "Green"  "Purple" "Red"    "Purple"
```

We could also access the `Color` column using `mice[["Color"]]`.

- We *replace* a column using the assignment operator. For example:

```
NewColor <- c("Blue", "White", "Orange", "Pink", "Magenta", "Blue", "Orange", "Blue")
mice$Color <- NewColor
```

Above, we could also have used any of the following:

```
mice[, 1] <- NewColor          # R will choose a name for the variable in mice
mice[[1]] <- NewColor          # R will choose a name for the variable in mice
mice[["Color"]] <- NewColor    # The variable in mice will be named Color
```

In the first two of these, R chooses a name for the variable in the data frame. In the third one, it uses `Color`.

6.2.3 Adding a New Column Using `cbind()`, `[ ]`, `[[ ]]`, or `$`

- We can add another column to a data frame using `cbind()`:

```
Bodyfat <- c(2.5, 2.1, 3.1, 3.0, 2.7, 2.6, 1.8, 2.0)
cbind(mice, Bodyfat)

##    Color Weight Length Bodyfat
## 1 Purple     23    3.8     2.5
## 2 Yellow     21    3.7     2.1
## 3    Red     12    3.0     3.1
## 4  Brown     26    3.4     3.0
## 5  Green     25    3.4     2.7
## 6 Purple     22    3.1     2.6
## 7    Red     26    3.5     1.8
## 8 Purple     19    3.2     2.0
```

Be aware that `cbind()` will convert a `"character"` vector to a *factor* unless you specify `stringsAsFactors = FALSE` in the call to `cbind()`.

- Alternatively, we can add a column to a data frame using `[ ]`, `[[ ]]`, or `$` and the assignment operator. For example, any of the following will work:

```
mice$Fat <- Bodyfat          # The variable in mice will be named Fat
mice[["Fat"]] <- Bodyfat     # The variable in mice will be named Fat
mice[[4]] <- Bodyfat         # R will choose a name for the variable in mice
mice[ , 4] <- Bodyfat        # R will choose a name for the variable in mice
```

  As per the comments, only the first two methods above allow you to choose a name for the new column. You'd have to add a name later (using `names()`) if you use the third or fourth one.

Section 6.2 Exercises

Exercise 2 Consider the following data set:

Status	Age	Education
Married	36	HS Diploma
Single	33	Bachelor of Arts
Single	21	Bachelor of Science
Married	29	Bachelor of Science
Single	19	HS Diploma
Married	35	Bachelor of Arts
Married	39	Master of Science
Single	28	HS Diploma
Single	21	HS Diploma

Here are vectors containing the data:

```
status <- c("Married", "Single", "Single", "Married", "Single", "Married",
            "Married", "Single", "Single")
educ <- c("HS Diploma", "Bachelor of Arts", "Bachelor of Science",
          "Bachelor of Science", "HS Diploma", "Bachelor of Arts",
          "Master of Science", "HS Diploma", "HS Diploma")
age <- c(36, 33, 21, 29, 19, 35, 39, 28, 21)
```

After creating the vectors, create a data frame containing the data:

```
my.data <- data.frame(Status = status, Age = age, Education = educ)
```

a) Write a command involving square brackets `[ , ]` and the assignment operator '`<-`' that replaces the age of the person in the first row of `my.data` by 37.

b) Write commands that obtains the entire first column of `my.data` using three different methods:

- Using single square brackets `[ , ]`.
- Using the dollar sign operator `$`.
- Using double square brackets `[[ ]]`.

c) Here's a vector:

```
age2 <- c(37, 23, 20, 39, 39, 25, 29, 18, 31)
```

Write commands that *replace* the entire second column of `my.data` by the `age2` vector using three different methods:

- Using single square brackets `[ , ]` and '<-'.
- Using the dollar sign operator `$` and '<-'.
- Using double square brackets `[[ ]]` and '<-'.

d) Here's another vector:

```
zipcode <- c(80204, 80217, 80110, 80223, 80204, 80117, 80004,
             80226, 80111)
```

Write commands that *add* the `zipcodes` as a new column of `my.data` using three different methods:

- Using single square brackets `[ , ]` and '<-'.
- Using the dollar sign operator `$` and '<-'.
- Using double square brackets `[[ ]]` and '<-'.

6.3 Viewing and Changing Variable Names in a Data Frame

- We can get or change the names of the columns of a data frame using:

```
names()          # Get or assign the names of the variables in a
                 # data frame
```

- For example:

```
names(mice)

## [1] "Color"  "Weight" "Length"

names(mice) <- c("C", "W", "L")
```

```
names(mice)

## [1] "C" "W" "L"
```

- To change just one of the column names, such as the 3rd one, we could use:

```
names(mice)[3] <- "Len"
```

Section 6.3 Exercises

Exercise 3 Create the following data frame:

```
x <- data.frame(A = 1:5, B = 6:10, C = c("a", "b", "c", "d", "e"))
```

a) Guess what the following command will return, then check your answer:

```
names(x)
```

b) Guess what the following command will do, then check your answer:

```
names(x) <- c("D", "E", "F")
```

c) Guess what the following command will do, then check your answer:

```
names(x)[3] <- "G"
```

6.4 Rearranging the Rows or Columns of a Data Frame

6.4.1 Rearranging the Rows or Columns

- We rearrange (permute) the rows (or columns) of a data frame using a vector of indices with the desired permutation before (or after) a comma in square brackets `[ ]`. For example:

```
my.data
```

```
##   Var1 Var2 Var3
## 1  Low    6   22
## 2  Low    3   18
## 3  Low   11   27
## 4  Med    9   27
## 5  Med   14   26
## 6  Med   15   29
## 7 High   19   28
## 8 High   18   31
## 9 High   21   30
```

```
my.data[ , c(2, 3, 1)]
```

```
##   Var2 Var3 Var1
## 1    6   22  Low
## 2    3   18  Low
## 3   11   27  Low
## 4    9   27  Med
## 5   14   26  Med
## 6   15   29  Med
## 7   19   28 High
## 8   18   31 High
## 9   21   30 High
```

```
my.data[9:1, ]
```

```
##   Var1 Var2 Var3
## 9 High   21   30
## 8 High   18   31
## 7 High   19   28
## 6  Med   15   29
## 5  Med   14   26
## 4  Med    9   27
## 3  Low   11   27
## 2  Low    3   18
## 1  Low    6   22
```

6.4.2 Sorting Rows by the Values of One Variable

- We can *sort* the rows of a data frame according to the values of one of its variables using square brackets `[ ]` and the `order()` function.
- `order()` takes a vector argument `x`, and returns a set of indices that will permute the elements of `x` in increasing order:

```
x <- c(7, 9, 5)
order(x)

## [1] 3 1 2
```

This says that the 3rd element of `x` is the smallest, the 1st element is the second smallest, and the 2nd element is the largest. Thus to sort the elements of `x`, we'd type:

```
x[c(3, 1, 2)]

## [1] 5 7 9
```

or just:

```
x[order(x)]

## [1] 5 7 9
```

- To sort the rows of `my.data` according to the values in `Var2`, we type:

```
my.data[order(my.data$Var2), ]

##   Var1 Var2 Var3
## 2  Low    3   18
## 1  Low    6   22
## 4  Med    9   27
## 3  Low   11   27
## 5  Med   14   26
## 6  Med   15   29
## 8 High   18   31
## 7 High   19   28
## 9 High   21   30
```

Notice that the rows of `my.data` remain intact, but they've been sorted according the values in `Var2`.

Section 6.4 Exercises

Exercise 4 Here's a small data frame:

```
x <- data.frame(x1 = c("c", "b", "a"), x2 = c(6, 4, 5),
                stringsAsFactors = FALSE)
x

##   x1 x2
## 1  c  6
## 2  b  4
## 3  a  5
```

a) Guess what the following command will return, then check your answer:

```
x[, c(2, 1)]
```

b) Guess what the following command will return, then check your answer:

```
x[c(3, 1, 2), ]
```

c) Guess what the following command will return, then check your answer:

```
x[order(x$x2), ]
```

d) What happens when you try to sort the rows of a data frame according to a `"character"` column? Try it with `x1` by typing:

```
x[order(x$x1), ]
```

6.5 Filtering on Data Frames

- ***Filtering*** a data frame means extracting rows for which some some condition is met. We can filter a data frame using either square brackets `[ ]` or `subset()`.

6.5.1 Filtering Rows Using Square Brackets `[ ]`

- We can filter rows of a data frame using a `"logical"` vector, whose `TRUE` elements indicate the rows to be extracted, before the comma in square brackets `[ ]`.

  For example, using `my.data` from above, the `"logical"` vector

```
my.data$Var1 == "Med"

## [1] FALSE FALSE FALSE  TRUE  TRUE  TRUE FALSE FALSE FALSE
```

  can be used to extract from `my.data` the rows for which `Var1` is `"Med"` as follows:

```
my.data[my.data$Var1 == "Med", ]

##   Var1 Var2 Var3
## 4  Med    9   27
## 5  Med   14   26
## 6  Med   15   29
```

6.5.2 Filtering Rows Using `subset()`

- `subset()` takes two main arguments, a data frame, `x`, and a `"logical"` vector, `subset`, whose `TRUE` elements indicate the rows to be extracted from `x`.

  For example, to extract from `my.data` the rows for which `Var1` is `"Med"`, type:

```
subset(my.data, subset = Var1 == "Med")

##   Var1 Var2 Var3
## 4  Med    9   27
## 5  Med   14   26
## 6  Med   15   29
```

  Notice that there's no need to use the dollar sign operator `$` with `Var1` in the call to `subset()`.

Section 6.5 Exercises

Exercise 5 The built-in R data frame `warpbreaks` contains data from a study of the strength of yarn used in weaving. There are three variables in the data set:

`breaks`	The number of breaks per loom, where a loom corresponds to a fixed length of yarn.
`wool`	The type of wool (A or B)
`tension`	The level of tension (L, M, H)

To look at the data, type:

```
warpbreaks
```

To read more about it, type :

```
? warpbreaks
```

a) Guess what will be returned by the following command, then check your answer.

```
warpbreaks[warpbreaks$tension == "M", ]
```

b) Guess what will be returned by the following command, then check your answer.

```
warpbreaks[warpbreaks$wool == "B", ]
```

c) Guess what will be returned by the following command, then check your answer.

```
subset(x = warpbreaks, subset = wool == "B")
```

d) Guess what will be returned by the following command, then check your answer.

```
subset(x = warpbreaks, subset = breaks < 15)
```

6.6 More on the Treatment of NAs

- If our data set contains NAs, we may want to exclude observations (rows) that contain NAs from the data analysis. The following function checks each row for NAs:

```
complete.cases()   # Checks each row of a data frame for completeness
                   # (i.e. no NAs), returning a "logical" vector
```

- `complete.cases()` takes a data frame as its main argument, and returns a `"logical"` vector whose elements are `TRUE` if the corresponding row of the data frame is "complete" (doesn't contain any NAs) and `FALSE` otherwise. Here's an example using a data frame `cars`:

```
cars
```

```
##          Make  Model Year CityMPG HighwayMPG
## 1        Ford  Focus 2012      17         25
## 2      Toyota  Prius 2013      51         NA
## 3        Ford Fusion 2014      22         31
## 4   Chevrolet   Volt 2015      NA         NA
## 5       Honda Accord 2011      23         33
## 6  Volkswagen Beetle 2012      22         31
## 7   Chevrolet Impala 2015      22         31
## 8       Tesla ModelS 2014      NA         NA
## 9       Honda  Civic 2011      26         34
## 10      Honda  S2000 2005      17         23
## 11     Toyota  Camry 2013      25         35
```

```
complete.cases(cars)
```

```
##  [1]  TRUE FALSE  TRUE FALSE  TRUE  TRUE  TRUE FALSE  TRUE  TRUE  TRUE
```

We use the `"logical"` vector returned by `complete.cases()` to obtain the data frame's "complete" rows:

```
cars[complete.cases(cars), ]
```

```
##          Make  Model Year CityMPG HighwayMPG
## 1        Ford  Focus 2012      17         25
## 3        Ford Fusion 2014      22         31
## 5       Honda Accord 2011      23         33
## 6  Volkswagen Beetle 2012      22         31
## 7   Chevrolet Impala 2015      22         31
## 9       Honda  Civic 2011      26         34
## 10      Honda  S2000 2005      17         23
## 11     Toyota  Camry 2013      25         35
```

Section 6.6 Exercises

Exercise 6 Here's a data frame `x`:

```
x <- data.frame(x1 = c(8, 2, 6, NA, 9, 4, 3),
                x2 = c("u", "r", "s", "v", "c", "t", "w"),
                x3 = c(2, NA, 4, 7, 7, NA, 1))
x
```

```
##   x1 x2 x3
## 1  8  u  2
## 2  2  r NA
## 3  6  s  4
## 4 NA  v  7
## 5  9  c  7
## 6  4  t NA
## 7  3  w  1
```

a) After creating the data frame `x`, guess what will be returned by the following command, then check your answer:

```
complete.cases(x)
```

b) Guess what will be returned by the following command, then check your answer:

```
x[complete.cases(x), ]
```

6.7 Merging Data Frames

6.7.1 Merging the Rows of Two Data Frames

- To merge (combine) the rows of two data frames, we can use `rbind()`:

```
rbind()       # Create a new data frame by "binding" the rows of one
              # data frame to those of another
```

- For example, we can combine the rows of the following two data frames:

```
NamesAndAges
```

```
##       Name Age
## 1     John  23
## 2    Karen  27
## 3 Margaret  19
## 4      Ann  36
## 5     Karl  32
## 6     Eric  24
```

```
MoreNamesAndAges
```

```
##     Name Age
## 1 Justin  22
## 2  Janet  28
```

by typing:

```
AllNamesAndAges <- rbind(NamesAndAges, MoreNamesAndAges)
```

The result is:

```
AllNamesAndAges

##       Name Age
## 1     John  23
## 2    Karen  27
## 3 Margaret  19
## 4      Ann  36
## 5     Karl  32
## 6     Eric  24
## 7   Justin  22
## 8    Janet  28
```

6.7.2 Merging the Columns of Two Data Frames

- To merge (combine) the columns of two data frames, some useful functions are:

```
cbind()       # Create a new data frame by "binding" the columns of one
              # data frame to those of another
merge()       # Merge two data frames that share one or more variables
              # (columns) in common
```

- For example, suppose we want to combine the following data frame with `NamesAndAges` (from above):

```
NamesAndWeights

##       Name Weight
## 1     John    155
## 2    Karen    170
## 3 Margaret    147
## 4      Ann    159
## 5     Karl    201
## 6     Eric    184
```

If we combine them using `cbind()`, we get:

```
cbind(NamesAndAges, NamesAndWeights)

##       Name Age     Name Weight
## 1     John  23     John    155
## 2    Karen  27    Karen    170
## 3 Margaret  19 Margaret    147
## 4      Ann  36      Ann    159
## 5     Karl  32     Karl    201
## 6     Eric  24     Eric    184
```

Notice that `Name` is common to both data frames, and `cbind()` leaves them as two separate columns. If the people were listed in different orders, `cbind()` would make no attempt to put them in the same order:

```
JumbledNamesAndWts
```

```
##       Name Weight
## 5     Karl    201
## 3 Margaret    147
## 1     John    155
## 6     Eric    184
## 2    Karen    170
## 4      Ann    159
```

```
cbind(NamesAndAges, JumbledNamesAndWts)
```

```
##       Name Age     Name Weight
## 5     John  23     Karl    201
## 3    Karen  27 Margaret    147
## 1 Margaret  19     John    155
## 6      Ann  36     Eric    184
## 2     Karl  32    Karen    170
## 4     Eric  24      Ann    159
```

We see that `cbind()` merges the data frames, but the `Ages` and `Weights` aren't in sync.

- To merge `NamesAndAges` with `JumbledNamesAndWts` so that the `Ages` and `Weights` are matched according to `Name`, use `merge()`:

```
merge(NamesAndAges, JumbledNamesAndWts, by = "Name")
```

```
##       Name Age Weight
## 1      Ann  36    159
## 2     Eric  24    184
## 3     John  23    155
## 4    Karen  27    170
## 5     Karl  32    201
## 6 Margaret  19    147
```

Note that:

  - `merge()` sorted the rows of both data frames alphabetically by `Name` before merging them.
  - Unlike `cbind()`, `merge()` didn't include a redundant `Name` column in the result.

- We can merge two data frames with matching done according to the values in *two* (or more) columns if we need to. For more information on this, see the help file for `merge()` (type `?merge`).

Section 6.7 Exercises

Exercise 7 Here are two data frames, `GroupA` and `GroupB`, containing responses to a survey question:

```
GroupA <- data.frame(RespondentID = c(1000, 1001, 1002, 1003),
                     Response = c(55, 62, 39, 45))
GroupA
```

```
##   RespondentID Response
## 1         1000       55
## 2         1001       62
## 3         1002       39
## 4         1003       45
```

```
GroupB <- data.frame(RespondentID = c(1004, 1005, 1006),
                     Response = c(70, 77, 56))
GroupB
```

```
##   RespondentID Response
## 1         1004       70
## 2         1005       77
## 3         1006       56
```

Guess what the following command returns, then check your answer:

```
rbind(GroupA, GroupB)
```

Exercise 8 Here are two data frames, `Question1` and `Question2`, containing responses by the *same* four people to *two different* survey questions:

```
Question1 <- data.frame(RespondentID = c(1000, 1001, 1002, 1003, 1004),
                        Response1 = c(55, 62, 39, 45, 70))
Question1
```

```
##   RespondentID Response1
## 1         1000        55
## 2         1001        62
## 3         1002        39
## 4         1003        45
## 5         1004        70
```

```
Question2 <- data.frame(RespondentID = c(1003, 1002, 1000, 1004, 1001),
                        Response2 = c(12, 17, 23, 24, 30))
Question2
```

```
##   RespondentID Response2
## 1         1003        12
## 2         1002        17
## 3         1000        23
## 4         1004        24
## 5         1001        30
```

Note that the `RespondentID`s are the same, but in different orders. We want to merge the columns of the two data frames. Which of the following commands would you recommend? Why?

```
cbind(Question1, Question2)
```

```
merge(Question1, Question2, by = "RespondentID")
```

6.8 Stacking and Unstacking Columns of a Data Frame

- Sometimes data are arranged in separate columns representing, say, different groups, but we'd prefer them to be in a single column with an adjacent column indicating the group.

  For example, the data might be "unstacked" like this:

```
unstacked.data

##   Grp1 Grp2 Grp3
## 1   23   19   31
## 2   11   26   28
## 3   14   24   34
## 4   16   29   25
```

  but we'd prefer them to be "stacked" like this:

```
stacked.data

##    Response Group
## 1        23  Grp1
## 2        11  Grp1
## 3        14  Grp1
## 4        16  Grp1
## 5        19  Grp2
## 6        26  Grp2
## 7        24  Grp2
## 8        29  Grp2
## 9        31  Grp3
## 10       28  Grp3
## 11       34  Grp3
## 12       25  Grp3
```

- Other times we need to do the opposite (take a set of "stacked" data and "unstack" it).
- The functions below are useful for such tasks.

```
stack()            # "Stack" columns of a data frame
unstack()          # "Unstack" a column of a data frame
```

- For example, using the `unstacked.data` from above, we can "stack" its columns by typing:

```
stacked.data <- stack(unstacked.data)
```

  and then add column names by typing:

```
names(stacked.data) <- c("Response", "Group")
```

  The result is the `stacked.data` shown above.

- To "unstack" a data set, we need indicate which column should be "unstacked" and which one indicates the groups.

We do this by passing a so-called ***formula*** to `unstack()` via its argument `form`. The *formula*'s left side is the variable to be "unstacked", and its right side is the group indicator.

For example, using `stacked.data` from above, to "unstack" it, we'd type:

```
unstacked.data <- unstack(stacked.data, form = Response ~ Group)
```

which produces the desired result:

```
unstacked.data

##   Grp1 Grp2 Grp3
## 1   23   19   31
## 2   11   26   28
## 3   14   24   34
## 4   16   29   25
```

(Above, the *formula* is written in R as `Response ~ Group`.)

Section 6.8 Exercises

Exercise 9

a) Here's a data frame containing data from an experiment involving a treatment group and a control group:

```
x <- data.frame(Ctrl = c(60, 44, 24, 56, 59),
                Trt = c(22, 45, 32, 45, 30))
x

##   Ctrl Trt
## 1   60  22
## 2   44  45
## 3   24  32
## 4   56  45
## 5   59  30
```

Guess what the following command will return, then check your answer:

```
stack(x)
```

b) Here's the same data, but in a different format:

```
z <- data.frame(Group = c("Trt", "Trt", "Trt", "Trt", "Trt", "Ctrl",
                          "Ctrl", "Ctrl", "Ctrl", "Ctrl"),
                Y = c(22, 45, 32, 45, 30, 60, 44, 24, 56, 59))
z
```

```
##    Group  Y
## 1    Trt 22
## 2    Trt 45
## 3    Trt 32
## 4    Trt 45
## 5    Trt 30
## 6   Ctrl 60
## 7   Ctrl 44
## 8   Ctrl 24
## 9   Ctrl 56
## 10  Ctrl 59
```

Guess what the following command will return, then check your answer:

```
unstack(z, form = Y ~ Group)
```